by
Jeff Prosise

# Tutor

## MEMORY ALLOCATION FUNCTIONS

Can you explain how the DOS functions that allocate and release blocks of memory work? I'm trying to write a short assembly language program to dynamically allocate (and then deallocate) memory. But when I execute the instructions

```
MOV   AH,48H
MOV   BX,0AH
INT   21H
```

to allocate 160 bytes, interrupt 21h returns with the carry flag set, indicating that my request for memory was denied. My system has 640K of RAM. What could be going wrong?

Peter Kirk
Glenfield, Auckland
New Zealand

- **MEMORY ALLOCATION FUNCTIONS: A look at the three functions a program can use to allocate, release, or resize a memory block.**
- **COMMAND PERFORMANCE: How to make sure the transient portion of COMMAND.COM gets reloaded.**

Function 48h—Allocate Memory—takes as an argument in BX the number of paragraphs of memory requested (a paragraph is 16 bytes). If the function returns with the carry flag clear, then the request was met and the segment address of the allocated block is returned in AX. If the request fails (usually because there was insufficient memory available to satisfy the request), the carry flag is set and BX returns with the size of the largest contiguous block available (in paragraphs). It's this feature that permits programs to determine how much RAM is free in the system. If function 48h is called with BX set to FFFFh (corresponding to 1MB of RAM, more conventional memory than can possibly be available), DOS returns the number of paragraphs free in BX.

Function 49h—Release Memory—reverses the action of function 48h. Once a block is allocated, it can be released—returned to the free memory pool—by calling function 49h with the segment address of the block in register ES. The segment addresses passed in ES must be the ones obtained earlier from function 48h. Blocks can be deallocated in any order, regardless of the order in which they were allocated.

Closely related to these functions, function 4Ah modifies the size of a memory block previously allocated with function 48h. You pass this function the segment address of the block to modify in ES and the new size of the block (again, in paragraphs) in BX. If the current structure of memory permits, DOS will shrink or expand the block to conform to the requested size and return with the carry flag clear. If for some reason the request can't be met, function 4Ah returns with carry set. Calls to this function rarely ever fail when a memory block is being reduced in size, but increasing the size of the block won't be possible if there's insufficient memory or if another block above the one being modified prevents it from being expanded to the desired size. Blocks of memory allocated with these functions are always contiguous.

When a program requests a block of memory, DOS begins in low memory

### HOW TO SHRINK A .COM FILE'S MEMORY ALLOCATION TO 64K

```
start:  mov  ah,4Ah     ;Function 4AH
        mov  bx,1000h   ;Block size = 4096 paragraphs
        int  21h
        jc   error      ;Branch on error
```

**Figure 1:** This code sequence, if executed by a program with a .COM extension and with ES pointing to the program's PSP segment, will reduce the amount of memory allocated to the program to 64K.

### FORMAT OF THE DOS MEMORY CONTROL BLOCK

| Offset(s) | Size | Description |
|---|---|---|
| 00h | Byte | End-of-chain identifier<br>4Dh = Not the last block<br>5Ah = Last block in chain |
| 01h | Word | PSP address of the program that owns this block of memory |
| 03h | Word | Length (in paragraphs) of this block of memory |
| 05h–07h | — — | Unused |
| 08h–0Fh | — — | ASCII name of the program installed in this block of memory if one is installed (DOS 4.*x* only; unused in earlier versions) |

**Figure 2:** Each block of memory is preceded by a 16-byte memory control block that specifies the size of the block, who owns it, and whether there are other blocks following it.

just above the area where the operating system kernel is loaded and searches upward until it finds an unused region of memory equal to or larger than the size of the requested block. Then it sets aside the amount of memory the program requested. If it reaches the top of memory without finding a suitable region of free memory, DOS returns an error code indicating that there's not enough memory.

You're probably having this problem because .COM files are allocated every byte of conventional memory available in the system when they're loaded. When your program asks for additional RAM (assuming it is a .COM file), DOS denies the request, because there's no memory left to allocate. .COM files, unlike .EXE files, don't have headers that define how much memory they need. To be safe, DOS allocates all remaining memory to a .COM program so that it's assured of having enough.

For good form, .COM programs should shrink the amount of memory they're allocated to just the amount they need when they first start up. Then subsequent requests for memory will be fulfilled, and other programs running in the system—TSRs, for example, which are free to make memory requests of their own—won't be starved for RAM.

It's easy for a .COM file to shrink its initial memory allocation down to 64K. The code in Figure 1 shows how it's done. Function 4Ah is called with the value 1000h in BX (1000h paragraphs is 64K) to shrink the block the program resides in. This routine assumes that the value in the ES register hasn't changed since the program was started. If it has, ES must be reloaded with the address of the segment where the program is loaded before interrupt 21h is called. After this code sequence is executed, your request to allocate an additional 160 bytes of memory should work.

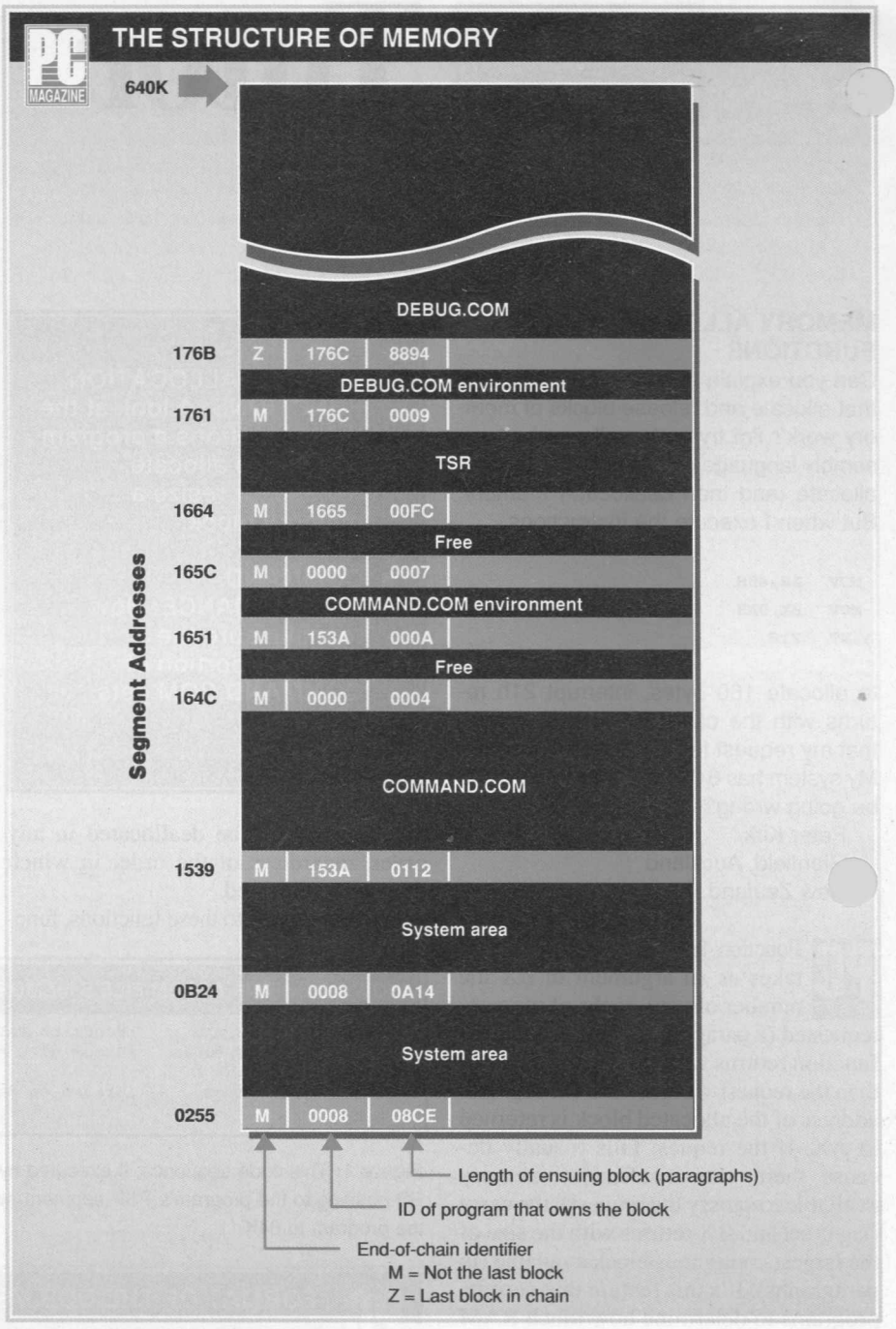


**Figure 3:** DOS divides memory into a series of blocks. Each block is preceded by a memory control block, or MCB, which specifies the size of the ensuing memory block, what owns it, and whether there are other blocks following this one. This memory map shows a typical layout for memory with one TSR installed and DEBUG.COM loaded and running. MCBs are highlighted in red.

### THE STRUCTURE OF MEMORY

To be really efficient with memory, you could shrink the memory block the program is loaded into even further if it doesn't require a full 64K. But remember that the SS register points to the base of the segment and that SP initially points to the top of the 64K segment, causing the stack to grow downward in memory from that point. If you shrink the memory block to less than 64K, you'll need to reset SP to the top of the resized block. Otherwise, the stack may be overwritten by other programs, resulting in a system crash. This won't happen under normal circumstances, because there's usually only one process running in the system at a time. But if a TSR grabs control of the CPU and requests more memory from DOS, it could be allocated the memory where the stack is located, if the proper precautions aren't taken. And that's Trouble, with a capital T.

Understanding how DOS structures memory also helps you understand DOS's

memory allocation and deallocation functions. DOS divides all the memory that it makes available to application programs into a series of blocks that can be as little as 1 paragraph in length or as large as 1MB. When a program is loaded, it is allocated two blocks: one to hold a local copy of the environment and one to store the program itself. If the program requests more memory, additional blocks are carved out of the remaining memory and set aside for it.

TSRs typically deallocate their own environment blocks before they terminate and become resident in memory. They don't usually need what's in the environment blocks, and releasing the memory makes more memory available for subsequent programs.

Each block is preceded in memory by a 16-byte *memory control block* (MCB), which specifies the size of the ensuing block, who owns it, and whether there are other blocks following it. The format of an MCB is diagrammed in Figure 2.

The first byte in the MCB indicates whether this is the last block in the chain. A value of 4Dh indicates that it isn't; a value of 5Ah indicates that it is. Not coincidentally, these values are actually the ASCII codes for the letters M and Z, the initials of one of Microsoft's primary DOS architects, Mark Zbikowski. The second and third bytes hold the program segment prefix (PSP) address of the program that owns the block.

A value of 0 indicates that the block is free; a nonzero value specifies that it's owned by somebody. In DOS 4.0, if the subsequent memory block holds a program, the name of the program is copied to the last 8 bytes of the MCB. These bytes were unused in the earlier versions of DOS.

The fourth and fifth bytes hold the length of the memory block (in paragraphs) that begins in the paragraph following the MCB. By adding the block length to the segment address of the current MCB and adding 1, you can derive the segment address of the next MCB in the chain. Together, the MCBs form a linked list that maps out the entire memory allocation chain.

Figure 3 illustrates how memory is structured on a typical PC with one TSR installed and DEBUG.COM loaded and running. MCBs are highlighted in red.

The first two memory blocks (whose MCBs are located at segments 0255h and 0B24h, respectively) are reserved for system use. The block following the MCB at segment 1539h holds COMMAND.COM. The blocks at segments 164Ch and 165Ch are free, while the block that separates them holds COMMAND.COM's copy of the environment. Note that the owner of the environment block is 153Ah, the segment where COMMAND.COM is loaded. The block at segment 1664h holds a TSR. The final two blocks hold DEBUG.COM (note that all the rest of memory is allocated to it) and its copy of the environment. When DEBUG terminates, all RAM above segment 1761h will be returned to the free memory pool to be used by subsequent programs.

**To conserve memory, COMMAND.COM splits itself into two halves when it's loaded: the resident portion, which sits in low memory; and the transient portion, which is loaded at the top of conventional memory.**

## COMMAND PERFORMANCE

When I exit certain programs, such as *Lotus 1-2-3,* my system hangs after displaying the message

```
Invalid COMMAND.COM
Cannot load COMMAND, system halted
```

COMMAND.COM is located in the \DOS directory of my C: drive, and my CONFIG.SYS file contains the statement

```
SHELL=C:\DOS\COMMAND.COM /E:512 /P
```

which clearly spells out the path to the command processor. What am I doing wrong?

Arthur Morgan
Grahamstown, South Africa

The mistake you're making is a common one: you're telling DOS where to find COMMAND.COM, but you're not telling *COMMAND.COM* where to find COMMAND.COM so that it can reload the transient portion of itself from disk when needed.

It may seem a little strange that COMMAND.COM needs to know what drive and directory it's located in, but there's a good reason. In order to conserve memory, COMMAND.COM splits itself into two halves when it's loaded: the *resident portion,* which sits in low memory; and the *transient portion,* which is loaded at the top of conventional memory. If an application program needs the space occupied by the transient part of COMMAND.COM, DOS lets the application have it. Then, when the program terminates and control goes back to COMMAND.COM, the resident portion of COMMAND.COM checks to see if the transient portion was corrupted and reloads it from disk if it was.

To find out where it's located on-d' COMMAND.COM looks at the COMSP environment string. If COMSPEC is set to C:\DOS\COMMAND.COM, for example, COMMAND.COM looks to the \DOS directory of drive C:. If there's no copy of COMMAND.COM there, the resident COMMAND.COM displays the "Invalid COMMAND.COM" error message you described and promptly halts the system.

There are two ways to correct the problem. One solution is to modify the SHELL statement in CONFIG.SYS to read

```
SHELL=C:\DOS\COMMAND.COM C:\DOS
 /E:512 /P
```

The extra string that's found between C:\DOS\COMMAND.COM and /E:512—C:\DOS—sets COMSPEC equal to C:\DOS\COMMAND.COM. In the absence of this parameter, COMSPEC is simply set to C:\COMMAND.COM, despite the fact that the SHELL directive loads COMMAND.COM from C:\DOS. With COMSPEC set this way, the system will lock up the first time COMMAND.COM is reloaded from disk if there isn't a y stored in the root directory.

A second way to correct the problem